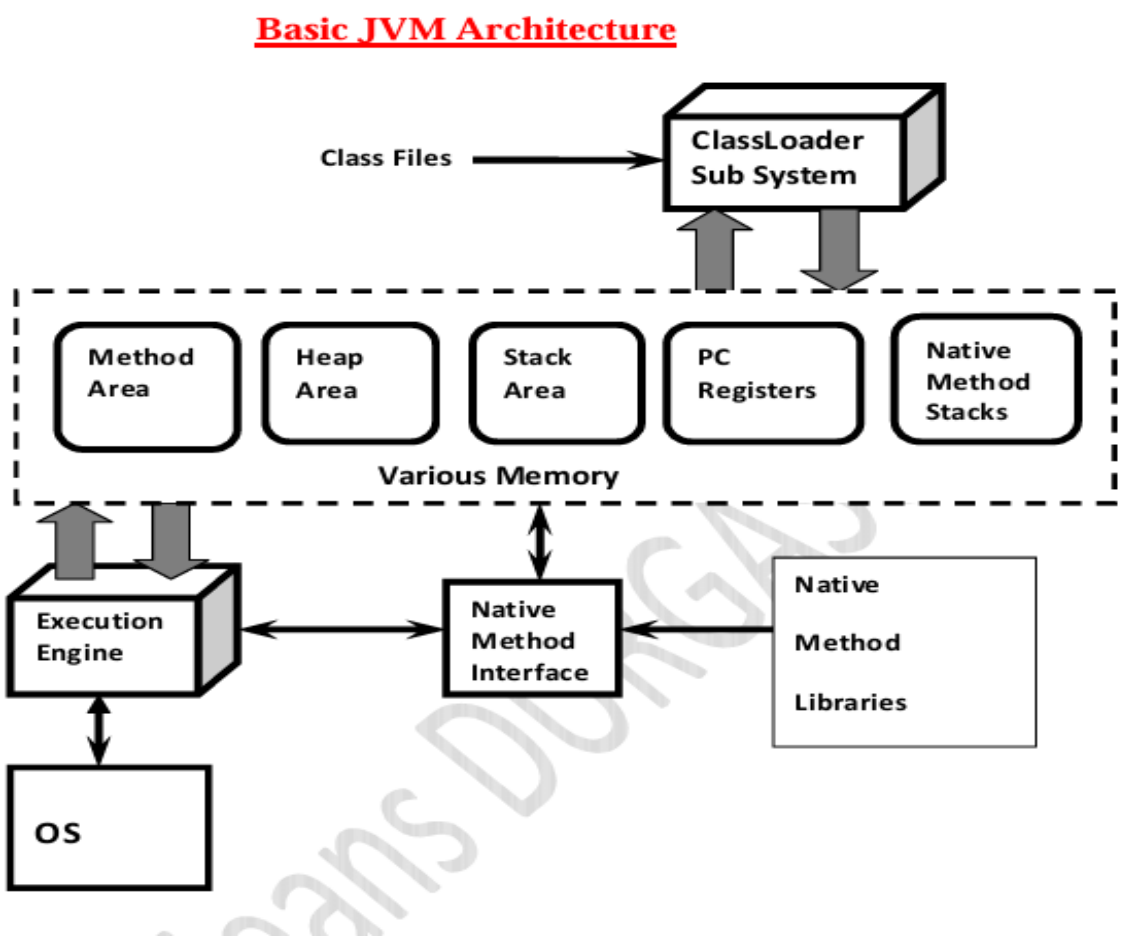

JVM (Java Virtual Machine) – Think of it like a smart box that runs Java programs.

1. CLASSLOADER SUBSYSTEM:

- What it does: Loads .class files (Java bytecode) into memory.
- Why it matters: It's the first step—without loading, nothing can run.

2. MEMORY AREAS:

- JVM divides memory into parts to manage different things:
- Method Area: Stores class-level info like method names and variables.
- Heap Area: Stores objects created in your program.
- Stack Area: Stores method calls and local variables.
- PC Registers: Keeps track of which instruction is being executed.
- Native Method Stack: Used when Java calls methods written in other languages like C/C++.

3. EXECUTION ENGINE:

- What it does: Actually, runs the bytecode instructions.
- How: It reads the bytecode and executes it step by step.

4. NATIVE METHOD INTERFACE:

- Purpose: Acts like a bridge between Java and other languages.
- Example: If Java needs to use a C function, this interface helps.

5. NATIVE METHOD LIBRARIES:

- What they are: Libraries written in other languages (like C).
- Used for: Performance or system-level tasks that Java alone can't do.

🧩 BREAKDOWN OF THE DIAGRAM:

- **ClassLoader**: Loads your Java code into memory.
- **Memory Areas**:
  - **Method Area**: Stores class info.
  - **Heap**: Stores objects.
  - **Stack**: Stores method calls and local variables.
  - **PC Register**: Tracks current instruction.
  - **Native Stack**: Helps run non-Java code.
- **Execution Engine**: Runs the code.
- **Native Interface & Libraries**: Connects Java to other languages like C/C++.

🔧 CLASSLOADER SUBSYSTEM:

It's a part of the JVM (Java Virtual Machine) that **loads your Java classes** into memory so they can be used in your program.

1️⃣ LOADING:

- It **reads the .class file** (compiled Java file) and stores its data in a special memory area called the **Method Area**.
- For each class, it stores:
  - Class name
  - Parent class name
  - Whether it's a class, interface, or enum
  - Modifiers (like public, abstract, etc.)
  - Variables and methods
  - Constant values
- After loading, JVM creates a special object called a **Class object** (of type Class) in the **Heap memory** to represent this class.

🧊 Memory Areas Involved

Area	Purpose
Method Area	Stores class-level data (metadata, static variables, method code)
Heap	Stores objects, including Class objects created by the ClassLoader

💉 Real-World Analogy

Think of the ClassLoader as a **librarian**:

- **Loading** = Fetching the book (class file)
- **Linking** = Checking if the book is valid and placing it on the shelf
- **Initialization** = Opening the book and reading the first page (static blocks)

🧠 **PRO TIPS FOR INTERVIEWS**

- JVM USES **LAZY LOADING** — CLASSES ARE LOADED ONLY WHEN NEEDED.
- YOU CAN CREATE YOUR OWN **CUSTOM CLASSLOADER** BY EXTENDING CLASSLOADER CLASS.
- CLASS.FORNAME("COM.EXAMPLE.MYCLASS") TRIGGERS **LOADING + LINKING + INITIALIZATION**.

2️⃣ LINKING PHASE:

Linking prepares the class for execution. It has **3 steps**:

1. VERIFICATION:

- JVM checks if the .class file is **valid and safe**.
- Uses **Bytecode Verifier** to ensure:
  - File is generated by a valid compiler.
  - No illegal bytecode or format issues.
- ❌ If it fails → java.lang.VerifyError

🧠 **Analogy**: Like airport security checking your ticket and baggage.

2. PREPARATION:

- JVM **allocates memory** for all **static variables**.
- Assigns **default values** (e.g., 0, null, false).
- **Original values** are assigned later during **Initialization**.

🧠 **Analogy**: Setting the table with empty plates — food comes later.

3. RESOLUTION:

- JVM replaces **symbolic references** (like class names) with **actual memory addresses**.
- Looks into the **Method Area** to find the real class definitions.

🧠 **Analogy**: Replacing contact names with actual phone numbers in your phone.

🔄 **Example:**

Java

```java
class Test {
    public static void main(String[] args) {
        String s = new String("Durga");
        Student s1 = new Student();
    }
}
```

- JVM loads: `Test.class`, `String.class`, `Student.class`, `Object.class`
- These names are stored in the **constant pool** of `Test.class`
- During **Resolution**, JVM links them to actual class definitions

3️⃣ INITIALIZATION PHASE:

- JVM:
  - Assigns **original values** to static variables
  - Executes **static blocks**
  - Order: **Top to bottom**, **Parent to Child**

🧠 **Analogy**: Powering on the machine — everything starts running.

🧠 JAVA CLASSLOADER TYPES:

The ClassLoader Subsystem has 3 main ClassLoaders:

1️⃣ BOOTSTRAP CLASSLOADER (A.K.A. PRIMORDIAL CLASSLOADER)

- Loads core Java classes (like String, Object, etc.)
- Loads from: jdk/jre/lib/rt.jar
- Built into the JVM (written in C/C++)
- You can’t access or extend it directly

🧠 Think of it as the "JVM's built-in loader" for essential Java classes.

2️⃣ EXTENSION CLASSLOADER

- Loads extension libraries
- Loads from: jdk/jre/lib/ext
- Child of Bootstrap ClassLoader
- Implemented in Java
- Class file: sun.misc.Launcher$ExtClassLoader

🧠 Think of it as the loader for optional Java features (like extra tools or APIs).

3️⃣ APPLICATION CLASSLOADER (A.K.A. SYSTEM CLASSLOADER)

- Loads your project classes (from your src or .class files)
- Loads from: Application Classpath (current working directory)
- Child of Extension ClassLoader
- Uses the CLASSPATH environment variable
- Class file: sun.misc.Launcher$AppClassLoader

🧠 Think of it as the loader for your own code.

🧠 JAVA CLASSLOADER DELEGATION HIERARCHY:

🔁 Delegation Principle

- Java ClassLoader follows the **Delegation Hierarchy Principle**.
- When the JVM encounters a class, it first checks if it's **already loaded** in the **Method Area**.
    - ✅ If loaded → JVM uses it.
    - ❌ If not loaded → JVM requests the **ClassLoader subsystem** to load it.

🧱 ClassLoader Subsystem Flow

1. **JVM → ClassLoader Subsystem**
2. **ClassLoader Subsystem → ApplicationClassLoader**
3. **ApplicationClassLoader → ExtensionClassLoader**
4. **ExtensionClassLoader → BootstrapClassLoader**

📂 Class Search Path

- **BootstrapClassLoader**
    - Searches in **Bootstrap Class Path**: jdk/jre/lib
    - If class found → loads it.
    - If not → delegates to **ExtensionClassLoader**
- **ExtensionClassLoader**
    - Searches in **Extension Class Path**: jdk/jre/lib/ext
    - If class found → loads it.
    - If not → delegates to **ApplicationClassLoader**
- **ApplicationClassLoader**
    - Searches in **Application Class Path**: Current Working Directory
    - If class found → loads it.
    - If not → throws:
        - ClassNotFoundException or
        - NoClassDefFoundError

🧠 JVM MEMORY AREAS:

1️⃣ METHOD AREA

- Method Area will be Created at the Time of JVM Start- Up.
- It will be Shared by All Threads (Global Memory).
- This Memory Area Need Not be Continuous.
- Method area shows runtime constant pool.
- Total Class Level Binary Information including Static Variables Stored in Method Area.

🧠 Think of it as: **"Class Info Storage"**

2️⃣ HEAP AREA:

- Programmer Point of View Heap Area is Consider as Important Memory Area.
- Heap Area will be Created at the Time of JVM Start- Up.
- Heap Area can be accessed by All Threads (Global OR Sharable Memory).
- Heap Area Nee Not be Continuous.
- All Objects and corresponding Instance Variables will be stored in the Heap Area.
- Every Array in Java is an Object and Hence Arrays Also will be stored in Heap Memory Only.

🧠 Think of it as: **"Object Storage"**

3️⃣ JAVA STACK AREA:

- **For Every Thread JVM will Create a Separate Runtime Stack.**
- **Runtime Stack will be Created Automatically at the Time of Thread Creation.**
- **All Method Calls and corresponding Local Variables, Intermediate Results will be stored in the Stack.**
- **For Every Method Call a Separate Entry will be Added to the Stack and that Entry is Called Stack Frame OR Activation Record.**
- **After completing that Method Call the corresponding Entry from the Stack will be Removed.**
- **After completing All Method Calls, Just Before terminating the Thread, the Runtime Stack will be destroyed by the JVM.**
- **The Data stored in the Stack can be accessed by Only the corresponding Thread and it is Not Available to Other Threads.**

🧠 **Think of it like a notebook for each thread — each method writes a page and removes it when done.**

📦 **STACK FRAME:**

Each method call creates a Stack Frame with 3 parts:

1️⃣ **Local Variable Array**

- **Stores method parameters and local variables.**
- **Each slot = 4 bytes**
- **Example:**
  - **int, float, Object → 1 slot**
  - **long, double → 2 slots**

2️⃣ **Operand Stack**

- **Used by JVM to do calculations.**
- **JVM instructions push and pop values here.**

🧠 **Think of it like a calculator memory — values go in and out during operations.**

3️⃣ **Frame Data**

- **Stores:**
  - **Symbolic references (like method info)**
  - **Exception handling info**

🧠 **Think of it like metadata — extra info to help JVM manage the method.**

4️⃣ PC REGISTER:

• For Every Thread a Separate PC Register will be Created at the Time of Thread Creation.

• PC Registers contains Address of Current executing Instruction.

• Once Instruction Execution Completes Automatically PC Register will be incremented to Hold Address of Next Instruction.

🧠 **Think of it as: "Instruction Pointer"**

5️⃣ NATIVE METHOD STACK:

• For Every Thread JVM will Create a Separate Native Method Stack.

• All Native Method (non-Java) Calls invoked by the Thread will be stored in the corresponding Native Method Stack.

🧠 Think of it as: **"Bridge to C/C++ code"**

📌 **Summary Table**

Memory Area	For JVM or Thread?	Stores What?
Method Area	JVM	Class info, static variables
Heap Area	JVM	Objects, instance variables
Stack Area	Thread	Method calls, local variables
PC Register	Thread	Current instruction address
Native Method Stack	Thread	Native method calls

🧠 **Quick Memory Placement:**

- **Static variables** → Method Area
- **Instance variables** → Heap
- **Local variables** → Stack

⚙️ JVM EXECUTION ENGINE:

- This is the Central Component of JVM.
- Execution Engine is Responsible to Execute Java Class Files.
- Execution Engine contains 2 Components for executing Java Classes.
    - Interpreter
    - JIT Compiler

🔹 INTERPRETER:

- It is Responsible to Read Byte Code and Interpret (Convert) into Machine Code (Native Code) and Execute that Machine Code Line by Line.
- The Problem with Interpreter is it Interpreters Every Time Even the Same Method Multiple Times. Which Reduces Performance of the System.
- To Overcome this Problem SUN People Introduced JIT Compilers in 1.1 Version.

🧠 **Think of it like reading instructions one step at a time, every time.**

🔹 JIT COMPILER (JUST-IN-TIME):

- **The Main Purpose of JIT Compiler is to Improve Performance.**
- **Internally JIT Compiler Maintains a Separate Count for Every Method whenever JVM Come Across any Method Call.**
- **First that Method will be interpreted normally by the Interpreter and JIT Compiler Increments the corresponding Count Variable.**
- **This Process will be continued for Every Method.**
- **Once if any Method Count Reaches Threshold (The Starting Point for a New State) Value, then JIT Compiler Identifies that Method Repeatedly used Method (HOT SPOT).**
- **Immediately JIT Compiler Compiles that Method and Generates the corresponding Native Code. Next Time JVM Come Across that Method Call then JVM Directly Use Native Code and Executes it Instead of interpreting Once Again. So that Performance of the System will be Improved.**
- **The Threshold Count Value varied from JVM to JVM.**
- **Some Advanced JIT Compilers will Re-compile generated Native Code if Count Reaches Threshold Value Second Time, So that More optimized Machine Code will be generated.**
- **Profiler which is the Part of JIT Compiler is Responsible to Identify HOT SPOTS.**

🧠 **Think of it like saving shortcuts for repeated tasks.**

🔍 Summary Table

Feature	Interpreter	JIT Compiler
Speed	Slower (line-by-line)	Faster (compiled hot methods)
Usage	All methods at least once	Only frequently used methods
Purpose	Basic execution	Performance optimization

🔗 JAVA NATIVE INTERFACE (JNI):

- JNI is a **bridge** between **Java code** and **native code** (like C or C++).
- It lets Java programs **call native methods** from external libraries.
- When you call hashCode() in Java, it might internally use native code via JNI.

🧠 **Think of JNI as a translator between Java and other languages.**

📦 JAVA CLASS FILE STRUCTURE:

1️⃣ Magic Number

- **First 4 bytes of the file.**
- **Always: 0xCAFEBABE**
- **✅ Tells JVM: "This is a valid Java class file."**
- **❌ If missing → JVM throws ClassFormatError**

2️⃣ Minor & Major Version

- **Shows which Java version compiled the file.**
- **Example:**
  - **JDK 1.4 → Major = 48**
  - **JDK 5 → Major = 49**
  - **JDK 6 → Major = 50**
- **❌ If JVM version is older → UnsupportedClassVersionError**

3️⃣ Constant Pool

- **Stores constants used in the class:**
  - **Method names**
  - **Class names**
  - **Strings**
  - **Numbers**
- **🧠 Think of it as a reference table for the class.**

4️⃣ Access Flags

- **Tells JVM about class properties:**
  - **Is it public?**
  - **Is it final?**
  - **Is it an interface?**

5️⃣ This Class

- **Points to the current class name in the constant pool.**

6️⃣ Super Class

- **Points to the parent class name in the constant pool.**

7️⃣ Interfaces

- **List of interfaces the class implements.**

8️⃣ Fields

- **Info about class variables (not local variables).**
- **Includes:**
  - **Name**
  - **Type**
  - **Access modifiers**

9 **Methods**

- **Info about methods in the class.**
- **Includes:**
  - **Name**
  - **Return type**
  - **Parameters**
  - **Bytecode instructions**

10 **Attributes**

- **Extra info like:**
  - **Source file name**
  - **Line number table**
  - **Annotations**